[19] 中华人民共和国国家知识产权局

[51] Int. Cl[7]
G06F 9/45
G06F 9/455 G06F 9/30

[12] 发明专利申请公开说明书

[21] 申请号 01812385.6

[43] 公开日 2003 年 9 月 3 日

[11] 公开号 CN 1440528A

[22] 申请日 2001.6.21 [21] 申请号 01812385.6
[30] 优先权
[32] 2000.10.5 [33] GB [31] 0024404.6
[86] 国际申请 PCT/GB01/02776 2001.6.21
[87] 国际公布 WO02/29563 英 2002.4.11
[85] 进入国家阶段日期 2003.1.6
[71] 申请人 ARM 有限公司
地址 英国剑桥郡
[72] 发明人 E·C·内维尔 A·C·罗斯

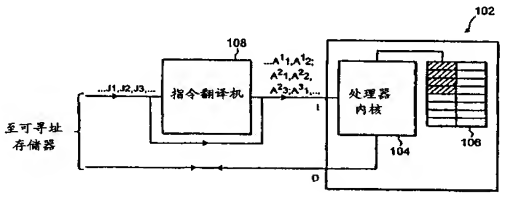
[74] 专利代理机构 中国专利代理(香港)有限公司
代理人 程天正 王 勇

权利要求书 3 页 说明书 27 页 附图 8 页

[54] 发明名称 寄存器中堆栈操作数的存储

[57] 摘要

一种数据处理装置(102)，它包含具有寄存器存储体(106)的处理器内核(104)。该寄存器存储体(106)包括一组用于存储堆栈操作数的寄存器。由指令翻译机(108)把来自第二指令集中指定堆栈操作数的指令翻译成第一指令集中指定寄存器操作数的指令(或者是与那些指令对应的控制信号)。然后由该处理器内核(104)执行这些已翻译的指令。该指令翻译机(108)具有多个变换状态，它们用于控制哪些寄存器与堆栈中的哪些堆栈操作数对应。变换状态之间变更的执行取决于向寄存器组添加堆栈操作数或从其中移出堆栈操作数。



ISSN1008-4274

1.用于处理数据的装置,所述装置包括:

配有包含多个寄存器的寄存器存储体的处理器内核,它运行时对保持在第一指令集中各指令中所指定的所述寄存器中的寄存器操作数执行操作;以及

一个指令翻译机,它在运行时把第二指令集中的指令翻译成与所述第一指令集中的指令对应的翻译机的输出信号,所述第二指令集中的各指令指定对保持在堆栈中的堆栈操作数要执行的各操作;其中

所述的指令翻译机运行时分配所述寄存器存储体中的寄存器组以便保持来自所述堆栈一部分的堆栈操作数;

所述指令翻译机具有多个变换状态,其中在所述寄存器组中的不同寄存器保持来自所述堆栈的所述部分中的不同位置的各堆栈操作数;并且

所述指令翻译机运行时根据包含在所述寄存器组中的堆栈操作数的填加或移出操作在各变换状态之间变更。

2.权利要求 1 中所记载的装置,其中所述翻译机的输出信号包括形成所述第一指令集中的指令的各信号。

3.在权利要求 1 和 2 的任何一个中所记载的装置,其中所述翻译机的输出信号包括控制所述处理器内核操作的控制信号以及在对所述第一指令集中的指令译码时所产生的匹配控制信号。

4. 在权利要求 1、2 和 3 的任何一个中所记载的装置,其中所述翻译机的输出信号包括,控制所述处理器内核的操作并指定在对所述第一指令集中的指令译码时所产生的控制信号没有指定的参数的控制信号。

5. 在前面的权利要求的任何一个中所记载的装置,其中所述指令翻译机提供变换状态,从而在不需要移动所述寄存器组中各寄存器之间的堆栈操作数的情况下,可把堆栈操作数填加到所述寄存器组或从所述寄存器组移出。

6.在前面的权利要求的任何一个中所记载的装置,其中所述寄存器组运行时,用于保持所述堆栈的栈顶位置的多个堆栈操作数,包括所述堆栈中的顶端位置的栈顶操作数。

7.在前面的权利要求的任何一个中所记载的装置,其中所述堆栈还包括多个保持堆栈操作数的可寻址存储器存储单元。

8. 在权利要求 7 中所记载的装置，其中堆栈操作数从所述寄存器组溢出进入所述多个可寻址存储器的存储单元。

9. 在权利要求 7 和 8 的任何一个中所记载的装置，其中保持在所述多个可寻址存储器存储单元的堆栈操作数在使用之前，要加载到所述寄存器组中。

10. 在前面的权利要求的任何一个中所记载的装置，其中所述指令翻译机使用多个指令模板，它们用于把来自所述第二指令集中的各指令翻译成来自第一指令集中的各指令。

11. 权利要求 10 中所记载的装置，其中来自所述第二指令集中的一个指令包括一个或多个堆栈操作数，它具有包括来自所述第一指令集的一个或多个指令的指令模板，在该模板中，把寄存器操作数变换成所述堆栈操作数。

12. 在前面的权利要求的任何一个中所记载的装置，其中所述指令翻译机包括如下的一个或多个:

硬件翻译逻辑;

解释用于控制计算机装置的程序代码的指令;

编译用于控制计算机装置的程序代码的指令; 以及

硬件编译逻辑。

13. 在前面的权利要求的任何一个中所记载的装置，其中所述指令翻译机包括这样的第一多个状态位，它们用于指示保持于所述寄存器组中的多个堆栈操作数。

14. 在权利要求 5 和 7-13 中的任何一个以及权利要求 6 中所记载的装置，其中所述指令翻译机包括这样的第二多个状态位，它们用于指示在所述寄存器组中的哪个寄存器保持所述栈顶操作数。

15. 在前面的权利要求的任何一个中所记载的装置，其中所述第二指令集是 Java 虚拟机指令集。

16. 一种处理数据的方法，该方法使用配有包含多个寄存器的寄存器存储体的处理器内核，它运行时对保持在第一指令集中各指令中所指定的所述寄存器中的寄存器操作数执行操作，所述方法包括步骤:

把第二指令集中的各指令翻译成对应所述第一指令集中的各指令的翻译机的输出信号，所述第二指令集中的各指令指定对保持在堆栈中的堆栈操作数要执行的操作;

分配所述寄存器存储体中的寄存器组以便保持来自所述堆栈部分的堆栈操作数;

采用多个转换状态中的这样的一种状态，在这种状态下，由所述寄存器组中的不同寄存器保持来自所述堆栈的所述部分中不同位置的各堆栈操作数；以及

根据对保持在所述寄存器组中的各堆栈操作数的填加或移出操作，变更各转换状态。

17.一种计算机程序产品,它包括用于控制计算机执行权利要求16的方法的计算机程序.

寄存器中堆栈操作数的存储

本发明涉及数据处理系统领域。更具体地讲，该发明涉及配有处理器内核的数据处理系统，其中该处理器内核具有寄存器存储体（bank）并且它执行与指令翻译机结合使用的第一指令集中的指令，该指令翻译机运行时把第二指令集的指令翻译成第一指令集的指令。

提供支持多个指令集的数据处理系统是公知的。这种系统的示例有英国 Cambridge 的 ARM 有限公司生产的 Thumb 使能处理器，它可以执行 16 位 Thumb 以及 32 位 ARM 这两种指令。Thumb 和 32 位 ARM 这两种指令执行的操作（例如，数学处理、加载、存储等）取决于指令中寄存器字段指定的并且存储于处理器内核的寄存器中的操作数。在开发能够有效利用处理器内核寄存器资源来生成迅速执行指令流的编译器方面花费了相当多的精力。

另一类指令集是这样的指令集，它们使用堆栈的方法来存储和处理要对其进行操作的操作数。在这种系统中的堆栈可以存储操作数值序列，这些值按照特定的序放置到堆栈中，并按照那个序的相反顺序把它们从堆栈中移出。因此，那个最后被放置到堆栈中的操作数通常也是首先移出堆栈的。基于堆栈的处理器可以提供这样的存储元件块，根据指示堆栈中当前“栈顶”位置的堆栈指针，堆栈操作数可以写入到该存储元件块还可以从该存储元件块中读取。堆栈指针指定堆栈存储器中的一个参考点，该点是要存入堆栈的最晚的堆栈操作数以及从该点可以参考对该堆栈的其它访问。同样地，在生产能够有效利用这种基于这种堆栈的处理器系统中的堆栈硬件资源的编译器上也花费了相当多的精力。

基于堆栈的指令集的专门示例是由 Sun Microsystems 有限公司所规定的 Java 虚拟机指令集。Java 编程语言试图提供一种这样的环境，在其中，用 Java 语言编写的计算机软件能够在多种不同处理硬件平台上执行，而不需要改变 Java 软件。

在数据处理系统中永远的目标就是：它们应该能够尽可能快速地执行控制它们的计算机软件。能够提高执行计算机软件速度的措施是极其需要的。

用于指令集和其它背景信息间翻译的已知系统的示例可以参见如下：US-A-5,805,895；US-A- 3,955,180；US-A-5,970,242；US-A-5,6 19,665；US-A-5,826,089；US-A-5,925, 123；US-A-5,875,33 6；US-A-5,937, 193；US-A-5,953,520；US-A-6,021,469；US-A-5,568,646；US-A-5,758,115；US-A-5,367,685；1998 年 3 月 IBM 技术公开报告中第 308-309 页的“用于精简指令集计算机的系统/370 模拟辅助处理器”；1986 年 7 月 IBM 技术公开报告中第 548-549 页的"全功能/1 指令集模拟器"；1994 年 3 月 IBM 技术公开报告中第 605-606 页的" RISC 处理器上的实时 CISC 体系结构 HW 模拟器"；1998 年 3 月 IBM 技术公开报告中第 272 页的"使用模拟控制块的性能改进"；1995 年 1 月 IBM 技术公开报告中第 537-540 页的"为在精简指令集计算机/循环系统上代码模拟的快速指令译码"；1993 年 2 月 IBM 技术公开报告中第 231-234 页的"高性能双体系结构处理器"；1989 年 8 月 IBM 技术公开报告中第 40-43 页的“系统/370 I/O 通道程序通道命令字的预取"；1985 年 6 月 IBM 技术公开报告中第 305-306 页的"全微码控制式模拟体系结构"；1972 年 3 月 IBM 技术公开报告中第 3074-3076 页的"用于模拟的操作码和状态处理"；1982 年 8 月 IBM 技术公开报告中第 954-956 页的"用大型系统最频繁使用的指令的微处理器的片内微编码和适合于编码剩余指令的基元"；1983 年 4 月 IBM 技术公开报告中第 5576-5577 页的"模拟指令"；S Furber 编著的书《ARM 系统体系结构》；Hennessy 和 Patterson 编著的书《计算机体系结构：定量方法》；以及 Tim Lindholm 和 Frank Yellin 编著的书《Java 虚拟机规范》第一版和第二版。

从本发明的一个方面来看，它提供用于处理数据的装置，所述装置包括：

配有包含多个寄存器的寄存器存储体的处理器内核，它运行时根据保持在第一指令集中各指令中所指定的所述寄存器中的寄存器操作数执行操作；以及

指令翻译机，用于在运行时把第二指令集中的指令翻译成与所述第一指令集中的指令对应的翻译机输出信号，所述第二指令集中的各指令根据保持在堆栈中的堆栈操作数指定要执行的各操作；其中

所述的指令翻译机运行时分配所述寄存器存储体中的寄存器组以

便保持来自所述堆栈部分的堆栈操作数;

所述指令翻译机具有多个变换状态，其中在所述寄存器组中的不同寄存器保持来自所述堆栈的所述部分中的不同位置的各堆栈操作数;并且

所述指令翻译机运行时根据包含在所述寄存器组中的堆栈操作数的填加或移出操作在各变换状态之间变更。

本发明提供第二、基于堆栈的指令集的指令的执行，该执行的过程是把这些指令翻译成在处理器内核上执行的第一、基于寄存器指令集的指令。本发明提供寄存器存储体内的一组寄存器以保持堆栈一部分的堆栈操作数。这样就有效地缓存处理器内核中的堆栈操作数以便加速执行。此外,为了更有效地使用已分配给堆栈操作数的各寄存器,指令翻译机有多个不同的变换状态，在各变换状态中，不同寄存器保持来自已缓存的堆栈部分内不同位置的堆栈操作数。以一种能提供与堆栈内的堆栈指针功能相似的方式的变换状态的变更，取决于对保持在堆栈所用寄存器内的堆栈操作数的填加或移出操作。这种方法会降低供应基于寄存器的处理器中的各寄存器内类似堆栈存储所需要的处理开销。

在本发明的优选实施例中，所述指令翻译机提供变换状态以便把堆栈操作数填加到所述寄存器组中或从其中移出，而不用在所述寄存器组内移动寄存器间的堆栈操作数。

这种优选特征如下：在试图提供一种这样的系统，在其中总是能在预先确定的寄存器中找到在堆栈中有具体位置的堆栈操作数的过程中，一旦把各堆栈操作数已存储于各寄存器中，使用各变换状态就可避免在寄存器间移动任何堆栈操作数，从而避免常常会出现的数量相当大的处理开销。

尽管可以理解，寄存器组能够保持来自堆栈内任何位置的堆栈操作数，但是人们还强烈希望寄存器组可以存储包含栈顶堆栈操作数的堆栈的栈顶部分。基于堆栈的处理系统多数情况下存取这样的堆栈操作数，这些操作数仅仅最近才存储于该堆栈，因此把这些堆栈操作数保持在它们可以被迅速存取到的寄存器内就具有强有力的优势。此外，把栈顶的堆栈操作数保持在各寄存器中可使指令翻译机具有这样的能力，当随着堆栈操作数在堆栈上的压入或弹出而使栈顶堆栈操作数常

常改变时，指令翻译机能极方便地在不同变换状态间变动，这一点极具优势。

尽管有可能给没有保持在寄存器内的堆栈部分提供各种不同的硬件装置，但是在本发明的优选实施例中，堆栈包括多个保持堆栈操作数的可寻址存储器存储单元。

可寻址存储器可频繁地在这样的具有各种机制的处理系统中见到，例如，包含有用于能够高速存取这种可寻址存储器内的数据的尖端高速缓存器。

可以理解，处理器内核的各寄存器可以用于存储堆栈操作数，但是这些寄存器却受到提供用于下述功能的其它那些寄存器需求的限制，这些功能例如是管理把来自第二指令集中的指令翻译成第一指令集，以及模拟诸如变量指针或常量池指针的其它控制值，其它那些寄存器可在基于堆栈的处理系统中见到。在这一场合，从为存储堆栈操作数提供的寄存器组中溢出的堆栈操作数可以保存在可寻址存储器中。

在一种补充方式中，许多基于高速寄存器的处理器系统被配置成提供数据处理操纵，该提供的数据处理操纵仅仅依赖于保持在寄存器内的数据值，以便避开由于相对长的存储器存取延迟等原因而引起的各种问题。在这样的场合，本发明提供在使用之前总是把堆栈操作数加载到寄存器组中。

指令翻译机可以方便地被配置成使用指令模板进行第二指令集和第一指令集间的翻译。由于可以实现第二指令集中的指令和通常第一指令集中的几个指令间转换这种特性，所以上述指令模板在一定程度上提供具有优势的灵活性。

可以理解，指令翻译机可以采用非常广泛的形式。具体地讲，所提供的指令翻译机可以作为用于翻译或编译第二指令集的专用硬件，或者作为控制处理器内核以实现与翻译或编译功能相似的软件。也可以有效地应用各方法的混合形式。在使用软件解释器的情况下，可以把翻译机的输出信号翻译成由软件解释器所生成的第一指令集的指令。

具体地讲，所提供的硬件翻译机可以用于实现对仅常常出现在第二指令集中的指令的高速翻译，而软件翻译机却可以用于第二指令集

中的复杂指令或者不怎么出现的指令，这样的指令要由硬件进行上述的翻译常常是不切实际的或者是低效率的。

一种可以控制指令翻译机转换状态的特别优选方式可提供：（1）指示保持在寄存器组中的堆栈操作数的数目的多个状态位；以及（2）指示哪个寄存器正在保持栈顶操作数的多个状态位。

尽管可以理解，第二指令集可以采用许多种不同的形式，但是在本发明的实施例中具体使用的是：第二指令集是 Java 虚拟机指令集。

从另一方面来看，本发明提供使用这样的处理器内核进行数据处理的方法，该处理器内核配有包含多个寄存器的寄存器存储体，并且它运行时可根据保持在第一指令集的指令中所指定的所述寄存器中寄存器操作数执行各操作，所述的方法包括如下的步骤：

把第二指令集中的指令翻译成与所述第一指令集中的指令对应的翻译机输出信号，所述第二指令集的指令指定根据保持在堆栈中的堆栈操作数要执行的各操作；

分配所述寄存器存储体中的寄存器组以便保持来自所述堆栈部分的堆栈操作数；

采用多个转换状态中的这样的一种状态，在这种状态下，由所述寄存器集中的不同寄存器保持来自所述堆栈的所述部分中不同位置的堆栈操作数；以及

根据对保持在所述寄存器组中的各堆栈操作数的填加或移出操作，在各转换状态间转变。

本发明也提供一种用于存储计算机程序的计算机程序产品，它根据上面所描述的技术来控制通用计算机。该计算机程序产品可以采用多种形式，例如软盘、光盘或者是从计算机网络上下载的计算机文件。

现在开始描述本发明的实施例，仅采用举例方式并参照伴随的附图，其中：

图 1 和 2 示意表示示例的指令流水线排列；

图 3 更详细地展示取阶段排列；

图 4 示意展示在取阶段从缓冲的指令字中读取变长的非本机指令；

图 5 示意展示一种数据处理系统，它用于执行处理器内核的本机指令和需要翻译的指令这两种；

图 6 示意展示，作为示范指令和状态序列，寄存器的内容，其用于堆栈操作数的存储、转换状态和需要翻译的指令与本机指令间的关系；

图 7 示意展示把非本机指令作为本机指令序列的执行；

图 8 展示这样的方式的流程示意图，使用这种方式，指令翻译机可以用一种给已翻译的指令保持中断延迟的方式操作；

图 9 示意展示使用软硬件技术把 Java 字节码翻译成 ARM 操作码；

图 10 示意展示基于硬件的翻译机、基于软件的翻译机以及基于软件的调度之间的控制流程图；

图 11 和 12 展示使用基于定时器方法的另一种控制调度操作的方式；以及

图 13 是信号示意图，它展示控制图 12 的电路操作的信号。

图 1 表示一种适用于基于 ARM 处理器的系统的类型的第一示例指令流水线 30。指令流水线 30 包括取阶段 32、本机指令(ARM/Thumb 指令)译码阶段 34、执行阶段 36、存储器存取阶段 38 以及回写阶段 40。执行阶段 36、存储器存取阶段 38 以及回写阶段 40 基本上是常规方式。在取阶段 32 以下、本机指令译码阶段 34 以上，提供有指令翻译机阶段 42。指令翻译机阶段 42 是一种有限状态机，它把 Java 可变长度的字节码指令翻译成本机的 ARM 指令。指令翻译机阶段 42 能够多步操作，从而使单个 Java 字节码指令可以产生这样的 ARM 指令序列，它们沿指令流水线 30 的其余部分馈送以便执行由 Java 字节码指令所指定的操作。简单 Java 字节码指令可以仅需要单个的 ARM 指令来执行它们的操作，而对于更复杂的 Java 字节码指令，或者在环境系统所规定的条件下，可能需要几个 ARM 指令来提供由 Java 字节码指令所指定的操作。这种多步骤操作发生在取阶段 32 以下，因此功率并不会消耗在取多个翻译的 ARM 指令或者来自存储器系统的 Java 字节码之上。Java 字节码指令按照常规方式存储在存储系统中，从而就不用在存储器系统上提供另外的约束，以便支持 Java 字节码的翻译操作。

如所展示的，给指令翻译机阶段 42 提供有旁路路径。当不使用指令翻译模式操作时，指令流水线 30 可以绕过指令翻译机阶段 42，并按照一种基本上没有变化的方式来操作而提供对本机指令的译码。

在指令流水线 30 中，指令翻译机阶段 42 是这样展示的：生成完

全代表对应 ARM 指令的翻译机输出信号，它通过多路复用器被传递到本机指令译码器 34。指令翻译机 42 也生成一些可以传递给本机指令译码器 34 的额外控制信号。在本机指令编码中的位空间约束可以对本机指令所指定的操作数范围施加限制。这些限制不需要与非本机指令共享。所提供的额外控制信号用于传递指定信号的附加指令，这些信号从非本机指令中推导出来，而不可能从存储于存储器内的那些本机指令中导出。作为示例，本机指令可能仅提供数量相对少的位用作本机指令范围内的立即操作数字段，而非本机指令可以允许有扩展的范围，通过使用额外的控制信号把立即操作数的扩展部分传递到翻译的本机指令之外的本机指令解码器 34，其中的本机指令也传递给本机指令译码器 34，这一点可以实现。

图 2 展示另一种指令流水线 44。在这个示例中，给系统提供有两个本机指令译码器 46、48 以及一个非本机指令译码器 50。非本机指令译码器 50 限制在它所能指定的各操作之内，这由提供用于支持本机指令的执行阶段 52、存储阶段 54 以及回写阶段 56 实现。因此，非本机指令译码器 50 必须有效地把非本机指令翻译成本机操作（它们可以是单个本机操作或者是本机操作序列），然后把适当的控制信号供应给执行阶段 52 以便实现这些一个或多个本机操作。可以理解，在这个例子中非本机指令译码器并不生成用于形成本机指令的那些信号，而是提供指定本机指令（或扩展本机指令）操作的那些控制信号。所生成的控制信号可能不匹配本机指令译码器 46、48 所生成的那些控制信号。

在操作中，由取阶段 58 所取的指令有选择地供应给指令译码器 46、48 或 50 之一，这要取决于使用展示的解复用器的特定处理模式。

图 3 更详细地原理展示指令流水线的取阶段。取指逻辑 60 从存储器系统中取定长指令字，然后把这些指令供应给指令字缓冲器 62。指令字缓冲器 62 是两侧摆动缓冲器，从而它可以存储当前的指令字和下一个指令字者这两者。每当当前指令字得到完全的译码时，译码就进展到下一个指令字，然后取逻辑 60 就用于把前一个当前的指令字替换为下一个要从存储器中取的指令字，即摆动缓冲器的每一侧会按照交叉方式增加两个连续存储的指令字。

在所展示的示例中，Java 字节码指令的最大指令长度是 3 个字

节。因此，要提供三个复用器，以便能够选择字缓冲器 62 任一侧中的任何 3 个相邻的字节，并把它们供应给指令翻译机 64。也给字缓冲器 62 和指令翻译机 64 提供在对本机指令进行取和译码时要用到的旁路路径 66。

可以看出，每个指令字是从存储器中一次取出的，然后存储于字缓冲器 62 中。当指令字翻译机 64 执行把 Java 字节码翻译成 ARM 指令时，单个指令字可以让多个 Java 字节码从其中读取。已翻译的可变长度本机指令序列的生成可以不需要多个存储系统读以及不需要消耗存储器资源，或者对存储系统施加其它限制，因为指令翻译操作限制在指令流水线上。

程序计数器值与正在翻译的每个 Java 字节码关联。这种程序计数器值沿着流水线的各阶段传递，从而在有必要时，使每个阶段能够使用关于其正在处理的特定 Java 字节码的信息。要翻译成多个 ARM 指令操作序列的 Java 字节码的程序计数器值并不增加，直到开始执行那个序列中的最后的 ARM 指令操作。以下述方式来保持程序计数器值会有利地简化本系统诸如调试和分支目标计算这样的其它方面，所述方式连续直接指向存储器中正在执行的指令。

图 4 原理性地展示从指令缓冲器 62 中对可变长度 Java 字节码指令的读操作。在第一阶段，具有长度为 1 的 Java 字节码得到读取和译码。在下一个阶段的 Java 字节码指令长度为 3 个字节，它横跨两个从存储器中读取的邻接指令字。这两个指令字存在于指令缓冲器 62 中，因此指令的译码和处理并不会受到由于在所取的指令间横跨了可变长度的指令而产生延迟。一旦 3 个 Java 字节码已经从指令缓冲器 62 中读取了，那么指令字中的较早取的那些指令字的重填就可以开始，因为对那些来自下面已经存在的指令中的 Java 字节码译码的后续处理继续进行。

在图 4 中所展示的最后阶段表示正被读取的第二个 3 字节码指令。这又一次横跨指令字。如果前面的指令字还没有完成它的重填，那么该指令的读可能会由于流水线的停顿而延迟，直到合适的指令字被存储到指令缓冲器 62 中为止。在一些实施方案中，定时可以是这样地，流水线从来不会由于这种类型的行为而停顿。可以理解，这个特定的例子出现的情况相对地较少，因为多数字节码的长度要比所展示

中的短，因此，两个在指令字间都横跨的连续译码相比较而言是不常见的。有效的信号可以采用下述方式与指令缓冲器 62 中的每个指令字相关联，即在已从其中读取了 Java 字节码之前，该方式能够发信号通知指令字是否已经被恰当地重填。

图 5 表示包括处理器内核 104 和寄存器存储体 106 的数据处理系统 102。给指令翻译机 108 提供指令路径，以便把 Java 虚拟机指令翻译成本机 ARM 指令（或者与其对应的控制信号），然后可以把它们供应给处理器内核 104。当正在从可寻址的存储器中取本机的 ARM 指令时，指令翻译机 108 可以进行旁路。可寻址的存储器可以是诸如配有另一个片外 RAM 存储器的高速缓存器这样的存储器系统。因为需要翻译的紧凑指令可以存储在存储器系统中，并且仅在紧接其传递到处理器内核 104 之前，可以扩张为本机指令，所以把指令翻译机 108 提供在存储器系统的下游，特别是高速缓存器下游允许有效地利用存储器系统的存储容量。

在这个示例中的寄存器存储体 106 包含 16 个通用的 32 位寄存器，其中的 4 个被分配用于存储堆栈操作数，即，存储堆栈操作数的寄存器组是寄存器 R0、R1、R2 以及 R3。

该寄存器组可以是空的、部分填充堆栈操作数的或者是全部填充堆栈操作数的。当前保持栈顶堆栈操作数的特定寄存器可以是该寄存器组中的任一寄存器。因此，可以理解，指令翻译机可以处于与一种状态对应的 17 个不同的变换状态中的任意一种状态，即各寄存器都是空的和四个组的四种状态，其中每种状态对应包含在寄存器组中的各自不同数目的堆栈操作数，并且用一个不同的寄存器保持栈顶操作数。表 1 展示为指令翻译机 108 的状态变换的 17 种不同的状态。可以理解，因为分配了不同数目的寄存器用于存储堆栈操作数，或者是因为特定处理器内核有其能够处理保持在寄存器中的数据值方式的约束，所以变换状态相当程度地取决于具体的实施方式，表 1 仅给出了一种具体实施的示例。

状态 00000			
R0 = 空			
R1 = 空			
R2 = 空			
R3 = 空			
状态 00100	状态 01000	状态 01100	状态 10000
R0 = TOS	R0 = TOS	R0 = TOS	R0 = TOS
R1 = 空	R1 = 空	R1 = 空	R1 = TOS-3
R2 = 空	R2 = 空	R2 = TOS-2	R2 = TOS-2
R3 = 空	R3 = TOS-1	R3 = TOS-1	R3 = TOS-1
状态 00101	状态 01001	状态 01101	状态 10001
R0 = 空	R0 = TOS-1	R0 = TOS-1	R0 = TOS-1
R1 = TOS	R1 = TOS	R1 = TOS	R1 = TOS
R2 = 空	R2 = 空	R2 = 空	R2 = TOS-3
R3 = 空	R3 = 空	R3 = TOS-2	R3 = TOS-2
状态 00110	状态 01010	状态 01110	状态 10010
R0 = 空	R0 = 空	R0 = TOS-2	R0 = TOS-2
R1 = 空	R1 = TOS-1	R1 = TOS-1	R1 = TOS-1
R2 = TOS	R2 = TOS	R2 = TOS	R2 = TOS
R3 = 空	R3 = 空	R3 = 空	R3 = TOS-3
状态 00111	状态 01011	状态 01111	状态 10011
R0 = 空	R0 = 空	R0 = 空	R0 = TOS-3
R1 = 空	R1 = 空	R1 = TOS-2	R1 = TOS-2
R2 = 空	R2 = TOS-1	R2 = TOS-1	R2 = TOS-1
R3 = TOS	R3 = TOS	R3 = TOS	R3 = TOS

表 1

在表 1 中，可以看出，状态值的开始 3 个位表示寄存器组中的非空寄存器的数目。状态值的最后两个位表示保持栈顶堆栈操作数的寄存器的寄存器数目。以这种方式，状态值可以很容易地用于控制硬件翻译机和软件翻译机的操作以便考虑寄存器组的当前占位和栈顶堆栈操作数的当前位置。

如图 5 展示的 Java 字节码 J1、 J2、 J3 的流从可寻址的存储器系统馈送到指令翻译机。接着指令翻译机 108 输出取决于输入的 Java 字节码的 ARM 指令（或等价的控制信号，可扩展）流、指令翻译机 8 的即时变换状态以及其它变量。所展示的示例表示把 Java 字节码 J1 变换成 ARM 指令 $A^1 1$ 和 $A^1 2$。把 Java 字节码 J2 变换成 ARM 指令 $A^2 1$、$A^2 2$ 和 $A^2 3$。最后，把 Java 字节码 J3 变换成 ARM 指令 $A^3 1$。每个 Java 字节

码可能会需要作为输入的一个或多个堆栈操作数，可以产生作为输出的一个或多个堆栈操作数。如果在本示例中的处理器内核 104 是具有加载/存储体系结构的 ARM 处理器内核，而在其中只有那些包含在寄存器中的数据值才可得到操纵，那么指令翻译机 108 被配置用于生成 ARM 指令，在把它们操纵或存储到可寻址存储器任何当前包含在寄存器组中的堆栈操作数之前，按照需要，把任何需要的堆栈操作数取到寄存器组中，以便给可能产生的结果堆栈操作数腾出空位。应当理解，可以认为每个 Java 字节码在执行前，都具有表示寄存器组中肯定存在的堆栈操作数的数目的一个关联的“需满”值，同时需要表示寄存器组中的空寄存器数目的"需空"值，该值在执行表示 Java 操作码的 ARM 指令之前必需获得。

表 2 展示了初始变换状态值、需满值、最终状态值以及关联的 ARM 指令间的关系。初始状态值和最终状态值对应表 1 中展示的变换状态。指令翻译机 108 确定与它正在翻译的特定 Java 字节码（操作码）关联的需满值。指令翻译机(108)执行 Java 字节码之前，它根据其所具有的初始变换状态确定是否需要把更多的堆栈操作数加载到寄存器组中。表 1 表示初始状态与被应用到 Java 字节码的需满值的测试，该值一起应用于使用关联的 ARM 指令（LDR 指令），以及使用堆栈高速缓存加载操作之后的最终变换状态来确定是否要把一个堆栈操作数加载到寄存器组中。实际上，如果在执行 Java 字节码之前，需要把不止一个堆栈操作数加载到寄存器组中，那么就会出现多种变换状态的过渡，每一个以一个关联的 ARM 指令把堆栈操作数加载到寄存器组中的寄存器之一。在一个不同的实施方案中，可以在单个状态过渡中加载多个堆栈操作数，因此可以改变到除了图 2 展示的那些状态以外的变换状态。

初始状态	需满	最终态	动作
00000	>0	00100	LDR R0, [Rstack, #-4]!
00100	>1	01000	LDR R3, [Rstack, #-4]!
01001	>2	01101	LDR R3, [Rstack, #-4]!
01110	>3	10010	LDR R3, [Rstack, #-4]!
01111	>3	10011	LDR R0, [Rstack, #-4]!
01100	>3	10000	LDR R1, [Rstack, #-4]!
01101	>3	10001	LDR R2, [Rstack, #-4]!
01010	>2	01110	LDR R0, [Rstack, #-4]!
01011	>2	01111	LDR R1, [Rstack, #-4]!
01000	>2	01100	LDR R2, [Rstack, #-4]!
00110	>1	01010	LDR R1, [Rstack, #-4]!
00111	>1	01011	LDR R2, [Rstack, #-4]!
00101	>1	01001	LDR R0, [Rstack, #-4]!

表 2

从表 2 可以看出，已加载到用于存储堆栈操作数的寄存器组中的一个新堆栈操作数会构成一个新的栈顶操作数，根据初始状态，这个新的栈顶操作数将会被加载到寄存器组中的一个特定的寄存器中。

表 3 以相似的方式展示初始状态、需空值、最终态以及关联的 ARM 指令间的关系，表示如果特定 Java 字节码的需空值指示在执行 Java 字节码之前，有必要给出初始状态，那么需把寄存器组中的一个寄存器清空以便在初始状态和最终态之间移动。用 STR 指令而保存到可寻址寄存器的特定寄存器值的变更要取决于哪个寄存器是当前的栈顶操作数。

初始状态	需空	最终态	动作
00100	>3	00000	STR R0, [Rstack], #4
01001	>2	00101	STR R0, [Rstack], #4
01110	>1	01010	STR R0, [Rstack], #4
10011	>0	01111	STR R0, [Rstack], #4
10000	>0	01100	STR R1, [Rstack], #4
10001	>0	01101	STR R2, [Rstack], #4
10010	>0	01110	STR R3, [Rstack], #4
01111	>1	01011	STR R1, [Rstack], #4
01100	>1	01000	STR R2, [Rstack], #4
01101	>1	01001	STR R3, [Rstack], #4
01010	>2	00110	STR R1, [Rstack], #4
01011	>2	00111	STR R2, [Rstack], #4
01000	>2	00100	STR R3, [Rstack], #4
00110	>3	00000	STR R2, [Rstack], #4
00111	>3	00000	STR R3, [Rstack], #4
00101	>3	00000	STR R1, [Rstack], #4

表 3

应当理解，在上面描述的示例系统中，需满和需空的条件是互斥

的，也就是说，在任意指定的时间对于指令翻译机正在尝试翻译的特定 Java 字节码来说，在需满或需空条件中仅有一个是真。由指令翻译机 108 所用的指令模板连同与其所选择的并由硬件指令翻译机 108 所支持的指令是这样选择的从而可使这种相互排斥需求得到满足。如果这种需求不适当，那么就可能出现这样的情况，特定字节码需要一些存在于寄存器组中的输入堆栈操作数，在执行代表 Java 字节码的指令之后，它们常常不允许获得足够的空寄存器，以便按照需要允许把执行的结果保持在寄存器中。

应当理解，给定的 Java 字节码会有整体净堆栈动作，它代表所消耗的堆栈操作数的数目以及在执行那个 Java 字节码时生成的堆栈操作数的数目之间的平衡。因为在执行之前，所消耗的堆栈操作数的数目是需要的，并且所生成的堆栈操作数的数目在执行之后也是需要的，所以在执行那个字节码之前，如果该净整体动作它本身会获得满足的情况下，与每个 Java 字节码关联的需满和需空值也必须要得到满足。表 4 展示初始状态、整体堆栈动作、最终态以及在寄存器应用和栈顶操作数（TOS）的相对位置的变化之间的关系。它可以是在实现表 4 展示的状态过渡之前所需要实现的一个或多个在表 2 或表 3 中所展示的状态过渡，以便根据 Java 字节码的需满和需空值为指定的 Java 字节码建立前提条件。

初始状态	堆栈动作	最终态	动作
00000	+1	00101	R1 <- TOS
00000	+2	01010	R1 <- TOS-1, R2 <- TOS
00000	+3	01111	R1 <- TOS-2, R2 <- TOS-1, R3 <- TOS
00000	+4	10000	R0 <- TOS, R1 <- TOS-3, R2 <- TOS-2, R3 <- TOS-1
00100	+1	01001	R1 <- TOS
00100	+2	01110	R1 <- TOS-1, R2 <- TOS
00100	+3	10011	R1 <- TOS-2, R2 <- TOS-1, R3 <- TOS
00100	-1	00000	R0 <- 空
01001	+1	01110	R2 <- TOS
01001	+2	10011	R2 <- TOS-1, R3 <- TOS
01001	-1	00100	R1 <- 空
01001	-2	00000	R0 <- 空 , R1 <- 空
01110	+1	10011	R3 <- TOS
01110	-1	01001	R2 <- 空
01110	-2	00100	R1 <- 空 , R2 <- 空
01110	-3	00000	R0 <- 空 , R1 <- 空 , R2 <- 空
10011	-1	01110	R3 <- 空
10011	-2	01001	R2 <- 空 , R3 <- 空
10011	-3	00100	R1 <- 空 , R2 <- 空 , R3 <- 空
10011	-4	00000	R0 <- 空 , R1 <- 空 , R2 <- 空 , R3<-空
10000	-1	01111	R0 <- 空
10000	-2	01010	R0 <- 空 , R3 <- 空
10000	-3	00101	R0 <- 空 , R2 <- 空 , R3 <- 空
10000	-4	00000	R0 <- 空 , R1 <- 空 , R2 <- 空 , R3 <-空
10001	-1	01100	R1 <- 空
10001	-2	01011	R0 <- 空 , R1 <- 空
10001	-3	00110	R0 <- 空 , R1 <- 空 , R3 <- 空
10001	-4	00000	R0 <- 空 , R1 <- 空 , R2 <- 空 , R3 <-空
10010	-1	01101	R2 <- 空

10010	-2	01000	R1 <- 空 , R2 <- 空
10010	-3	00111	R0 <- 空 , R1 <- 空 , R2 <- 空
10010	-4	00000	R0 <- 空 , R1 <- 空 , R2 <- 空 , R3 <- 空
01111	+1	10000	R0 <- TOS
01111	-1	01010	R3 <- 空
01111	-2	00101	R2 <- 空 , R3 <- 空
01111	-3	00000	R1 <- 空 , R2 <- 空 , R3 <- 空
01100	+1	10001	R1 <- TOS
01100	-1	01011	R0 <- 空
01100	-2	00110	R0 <- 空 , R3 <- 空
01100	-3	00000	R0 <- 空 , R2 <- 空 , R3 <- 空
01101	+1	10010	R2 <- TOS
01101	-1	01000	R1 <- 空
01101	-2	00111	R0 <- 空 , R1 <- 空
01101	-3	00000	R0 <- 空 , R1 <- 空 , R3 <- 空
01010	+1	01111	R3 <- TOS
01010	+2	10000	R3 <- TOS-1, R0 <- TOS
01010	-1	00101	R2 <- 空
01010	-2	00000	R1 <- 空 , R2 <- 空
01011	+1	01100	R0 <- TOS
01011	+2	10001	R0 <- TOS-1, R1 <- TOS
01011	-1	00110	R3 <- 空
01011	-2	00000	R2 <- 空 , R3 <- 空
01000	+1	01101	R1 <- TOS
01000	+2	10010	R1 <- TOS-1, R2 <- TOS
01000	-1	00111	R0 <- 空
01000	-2	00000	R0 <- 空 , R3 <- 空
00110	+1	01011	R3 <- TOS
00110	+2	01100	R0 <- TOS, R3 <- TOS-1
00110	+3	10001	R1 <- TOS, R0 <- TOS-1, R3 <- TOS-2
00110	-1	00000	R2 <- 空
00111	+1	01000	R0 <- TOS
00111	+2	01101	R0 <- TOS-1, R1 <- TOS
00111	+3	10010	R0 <- TOS-2, R1 <- TOS-1, R2 <- TOS
00111	-1	00000	R3 <- 空
00101	+1	01010	R2 <- TOS
00101	+2	01111	R2 <- TOS-1, R3 <- TOS
00101	+3	10000	R2 <- TOS-2, R3 <- TOS-1, R1 <- TOS
00101	-1	00000	R1 <- 空

表 4

可以理解，表 2、表 3 和表 4 展示的状态和条件间的各关系可以合并成单一的状态过渡表或者状态示意图，但是它们已经在上面分别进行了展示以便辅助说明。

不同状态、条件以及净动作间的关系可以用来定义控制指令翻译

机 108 的这一操作方面的硬件状态机（采用有限状态机的形式）。备选地，这些关系可以通过软件或者硬件与软件的组合来建立模型。

下面是这样的可能的 Java 字节码子集的示例，它为该子集中的每个 Java 字节码指示所关联的需满、需空以及堆栈动作的值，这些值用于可以结合表 2、3 和 4 使用的那个字节码。

```
--- iconst_0

操作:          压入整型常量

堆栈:          ... =>
               ..., 0

               Require-Full = 0
               Require-Empty = 1
               Stack-Action = +1

--- iadd

操作:          加整型数

堆栈:          ..., value1, value2 =>
               ..., result

               Require-Full = 2
               Require-Empty = 0
               Stack-Action = -1

--- lload_0

操作:          从局部变量加载long型

堆栈:          ... =>
               ..., value.word1, value.word2

               Require-Full = 0
               Require-Empty = 2
               Stack-Action = +2

--- lastore
```

```
操作:              存储到long型数组
堆栈:              ..., arrayref, index, value.word1, value.word2 =>
                   ...

                   Require-Full = 4
                   Require-Empty = 0
                   Stack-Action = -4

--- land

操作:              Boolean AND long
堆栈:              ..., value1.word1, value1.word2, value2.word1,
value2.word2 =>
                   ..., result.word1, result.word2

                   Require-Full = 4
                   Require-Empty = 0
                   Stack-Action = -2

--- iastore

操作:              存储到数组
堆栈:              ..., arrayref, index, value =>
                   ...

                   Require-Full = 3
                   Require-Empty = 0
                   Stack-Action = -3

--- ineg

操作:              Negate int
堆栈:              ..., value =>
                   ..., result

                   Require-Full = 1
                   Require-Empty = 0
                   Stack-Action = 0
```

下面还有用于上面所叙述的每个 Java 字节码指令的指令模板示例。所表示的指令是实施每个 Java 字节码所需要的行为的 ARM 指令。寄存器字段"TOS-3"、"TOS-2"、"TOS-1"、"TOS"、"TOS+1"以及"TQS+2"可以根据当时所采用的转换状态使用如从表 1 中所读取的适当的寄存器指定符替换。指示"TQS+n"表示在当前用于存储栈顶操作数的寄存器之上的第 N 个寄存器，它从存储栈顶操作数的寄存器开始并用寄存器的值向上计数，直到达到寄存器组的末端为止，在该点有一个包指向寄存器组中的第一个寄存器。

```
iconst_0        MOV     tos+1, #0

lload_0         LDR     tos+2, [vars, #4]
                LDR     tos+1, [vars, #0]

iastore         LDR     Rtmp2, [tos-2, #4]
                LDR     Rtmp1, [tos-2, #0]
                CMP     tos-1, Rtmp2, LSR #5
                BLXCS   Rexc
                STR     tos, [Rtmp1, tos-1, LSL #2]

lastore         LDR     Rtmp2, [tos-3, #4]
                LDR     Rtmp1, [tos-3, #0]
                CMP     tos-2, Rtmp2, LSR #5
                BLXCS   Rexc
                STR     tos-1, [Rtmp1, tos-2, LSL #3]!
                STR     tos, [Rtmp1, #4]

iadd            ADD     tos-1, tos-1, tos

ineg            RSB     tos, tos, #0

land            AND     tos-2, tos-2, tos
                AND     tos-3, tos-3, tos-1
```

按照上面描述的技术，在下面给出了由硬件翻译单元 108 所执行的单个 Java 字节码的执行序列的示例。执行序列表示：随着各指令的执行，初始状态进展通过状态序列，生成作为每种状态过渡中执行动作的结果的 ARM 指令序列，整个过程实现了把 Java 字节码翻译成 ARM 指令序列。

```
初始态:          00000
指令:            iadd (Require-Full=2, Require-Empty=0, Stack-Action=-1)

条件:          Require-Full>0
状态转变:        00000      >0     00100
ARM指令(S):
                                LDR R0, [Rstack, #-4]!
下一个状态:    00100
指令:            iadd (Require-Full=2, Require-Empty=0, Stack-Action=-1)

条件:          Requite-Full>1
状态转变:        00100      >1     01000
ARM指令(S):
```

```
                                        LDR R3, [Rstack, #-4]!
下一个状态:      01000
指令:                iadd (Require-Full=2, Require-Empty=0, Stack- Action=-1)

条件:            Stack-Action=-1
状态转变:          01000       -1      00111
指令模板:
                  ADD    tos-1, tos-1, tos
ARM指令(S)(替换后):
                                        ADD R3, R3, R0
下一个状态:      00111
```

图 6 展示使用不同的方式对一些另外的 Java 字节码指令的执行。图 6 的顶部展示的是 ARM 指令序列、变换状态的变更以及在执行 iadd Java 字节码指令时所出现的寄存器内容。初始变换状态 00000 对应寄存器组内的所有寄存器都是空的情况。生成的头两个 ARM 指令用于把两个堆栈操作数弹出到栈顶"TOS"寄存器是 R0 并用于存储堆栈操作数的寄存器中。第三个 ARM 指令实际上执行加法操作并把结果写入寄存器 R3 中(它现在成为了栈顶操作数),同时消耗以前保持在寄存器 R1 中的堆栈操作数,因此,产生 1 的整体堆栈动作。

接着处理进展到执行两个 Java 字节码,其中每个代表两个堆栈操作数的长加载。用于第一 Java 字节码的 2 的需空条件立即得到满足,因此两个 ARM LDR 指令可以被发布并得到执行。在执行 Java 字节码的第一长加载之后,变换状态是 01101。在这个状态中,寄存器组仅包含单个空寄存器。第二个 Java 字节码的长加载指令具有得不到满足的 2 的需空值,因此所需要的第一活动就使用 ARM STR 指令把堆栈操作数压入到可寻址的存储器。这就释放了寄存器组中的一个寄存器,它可供有可能作为两个下面的 LDR 指令要加载的部分的一个新的堆栈操作数使用。如前面所提到的,指令翻译可用硬件、软件以及这两者的组合实现。下面给出的是按照上面所描述的技术而生成的示例软件解释器的一个子部分。

```
Interpret               LDRB    Rtmp, [Rjpc, #1]!
                        LDR     pc, [pc, Rtmp, lsl #2]
                        DCD     0
                        ...
                        DCD     do_iconst_0      ; Opcode 0x03
                        ...
                        DCD     do_lload_0       ; Opcode 0x1e
                        ...
                        DCD     do_iastore       ; Opcode 0x4f
                        DCD     do_lastore       ; Opcode 0x50
                        ...
                        DCD     do_iadd          ; Opcode 0x60
                        ...
                        DCD     do_ineg          ; Opcode 0x74
                        ...
                        DCD     do_land          ; Opcode 0x7f
                        ...
do_iconst_0             MOV     R0, #0
                        STR     R0, [Rstack], #4
                        B       Interpret
do_lload_0              LDMIA   Rvars, {R0, R1}
                        STMIA   Rstack!, {R0, R1}
                        B       Interpret
do_iastore              LDMDB   Rstack!, {R0, R1, R2}
                        LDR     Rtmp2, [r0, #4]
                        LDR     Rtmp1, [r0, #0]
                        CMP     R1, Rtmp2, LSR #5
                        BCS     ArrayBoundException
                        STR     R2, [Rtmp1, R1, LSL #2]
                        B       Interpret
do_lastore              LDMDB   Rstack!, {R0, R1, R2, R3}
                        LDR     Rtmp2, [r0, #4]
                        LDR     Rtmp1, [r0, #0]
                        CMP     R1, Rtmp2, LSR #5
                        BCS     ArrayBoundException
                        STR     R2, [Rtmp1, R1, LSL #3]!
                        STR     R3, [Rtmp1, #4]
                        B       Interpret
do_iadd                 LDMDB   Rstack!, {r0, r1}
                        ADD     r0, r0, r1
                        STR     r0, [Rstack], #4
                        B       Interpret
do_ineg                 LDR     r0, [Rstack, #-4]!
                        RSB     tos, tos, #0
                        STR     r0, [Rstack], #4
                        B       Interpret
do_land                 LDMDB   Rstack!, {r0, r1, r2, r3}
                        AND     r1, r1, r3
                        AND     r0, r0, r2
                        STMIA   Rstack!, {r0, r1}
                        B       Interpret

State_00000_Interpret   LDRB    Rtmp, [Rjpc, #1]!
                        LDR     pc, [pc, Rtmp, lsl #2]
                        DCD     0
                        ...
                        DCD     State_00000_do_iconst_0 ; Opcode 0x03
                        ...
```

```
                        DCD     State_00000_do_lload_0  ; Opcode 0x1e
                        ...
                        DCD     State_00000_do_iastore  ; Opcode 0x4f
                        DCD     State_00000_do_lastore  ; Opcode 0x50
                        ...
                        DCD     State_00000_do_iadd     ; Opcode 0x60
                        ...
                        DCD     State_00000_do_ineg     ; Opcode 0x74
                        ...
                        DCD     State_00000_do_land     ; Opcode 0x7f
                        ...
State_00000_do_iconst_0 MOV     R1, #0
                        B       State_00101_Interpret
State_00000_do_lload_0  LDMIA   Rvars, {R1, R2}
                        B       State_01010_Interpret
State_00000_do_iastore  LDMDB   Rstack!, {R0, R1, R2}
                        LDR     Rtmp2, [r0, #4]
                        LDR     Rtmp1, [r0, #0]
                        CMP     R1, Rtmp2, LSR #5
                        BCS     ArrayBoundException
                        STR     R2, [Rtmp1, R1, LSL #2]
                        B       State_00000_Interpret
State_00000_do_lastore  LDMDB   Rstack!, {R0, R1, R2, R3}
                        LDR     Rtmp2, [r0, #4]
                        LDR     Rtmp1, [r0, #0]
                        CMP     R1, Rtmp2, LSR #5
                        BCS     ArrayBoundException
                        STR     R2, [Rtmp1, R1, LSL #3]!
                        STR     R3, [Rtmp1, #4]
                        B       State_00000_Interpret
State_00000_do_iadd     LDMDB   Rstack!, {R1, R2}
                        ADD     r1, r1, r2
                        B       State_00101_Interpret
State_00000_do_ineg     LDR     r1, [Rstack, #-4]!
                        RSB     r1, r1, #0
                        B       State_00101_Interpret
State_00000_do_land     LDR     r0, [Rstack, #-4]!
                        LDMDB   Rstack!, {r1, r2, r3}
                        AND     r2, r2, r0
                        AND     r1, r1, r3
                        B       State_01010_Interpret

State_00100_Interpret   LDRB    Rtmp, [Rjpc, #1]!
                        LDR     pc, [pc, Rtmp, lsl #2]
                        DCD     0
                        ...
                        DCD     State_00100_do_iconst_0 ; Opcode 0x03
                        ...
                        DCD     State_00100_do_lload_0  ; Opcode 0x1e
                        ...
                        DCD     State_00100_do_iastore  ; Opcode 0x4f
                        DCD     State_00100_do_lastore  ; Opcode 0x50
                        ...
                        DCD     State_00100_do_iadd     ; Opcode 0x60
                        ...
                        DCD     State_00100_do_ineg     ; Opcode 0x74
                        ...
                        DCD     State_00100_do_land     ; Opcode 0x7f
                        ...
State_00100_do_iconst_0 MOV     R1, #0
```

```
                        B       State_01001_Interpret
State_00100_do_lload_0  LDMIA   Rvars, {r1, R2}
                        B       State_01110_Interpret
State_00100_do_iastore  LDMDB   Rstack!, {r2, r3}
                        LDR     Rtmp2, [r2, #4]
                        LDR     Rtmp1, [r2, #0]
                        CMP     R3, Rtmp2, LSR #5
                        BCS     ArrayBoundException
                        STR     R0, [Rtmp1, R3, lsl #2]
                        B       State_00000_Interpret
State_00100_do_lastore  LDMDB   Rstack!, {r1, r2, r3}
                        LDR     Rtmp2, [r1, #4]
                        LDR     Rtmp1, [r1, #0]
                        CMP     r2, Rtmp2, LSR #5
                        BCS     ArrayBoundException
                        STR     r3, [Rtmp1, r2, lsl #3]!
                        STR     r0, [Rtmp1, #4]
                        B       State_00000_Interpret
State_00100_do_iadd     LDR     r3, [Rstack, #-4]!
                        ADD     r3, r3, r0
                        B       State_00111_Interpret
State_00100_do_ineg     RSB     r0, r0, #0
                        B       State_00100_Interpret
State_00100_do_land     LDMDB   Rstack!, {r1, r2, r3}
                        AND     r2, r2, r0
                        AND     r1, r1, r3
                        B       State_01010_Interpret

State_01000_Interpret   LDRB    Rtmp, [Rjpc, #1]!
                        LDR     pc, [pc, Rtmp, lsl #2]
                        DCD     0
                        ...
                        DCD     State_01000_do_iconst_0 ; Opcode 0x03
                        ...
                        DCD     State_01000_do_lload_0  ; Opcode 0x1e
                        ...
                        DCD     State_01000_do_iastore  ; Opcode 0x4f
                        DCD     State_01000_do_lastore  ; Opcode 0x50
                        ...
                        DCD     State_01000_do_iadd     ; Opcode 0x60
                        ...
                        DCD     State_01000_do_ineg     ; Opcode 0x74
                        ...
                        DCD     State_01000_do_land     ; Opcode 0x7f
                        ...
State_01000_do_iconst_0 MOV     R1, #0
                        B       State_01101_Interpret
State_01000_do_lload_0  LDMIA   Rvars, {r1, r2}
                        B       State_10010_Interpret
State_01000_do_iastore  LDR     r1, [Rstack, #-4]!
                        LDR     Rtmp2, [R3, #4]
                        LDR     Rtmp1, [R3, #0]
                        CMP     r0, Rtmp2, LSR #5
                        BCS     ArrayBoundException
                        STR     r1, [Rtmp1, r0, lsl #2]
                        B       State_00000_Interpret
State_01000_do_lastore  LDMDB   Rstack!, {r1, r2}
                        LDR     Rtmp2, {r3, #4}
                        LDR     Rtmp1, {R3, #0}
                        CMP     r0, Rtmp2, LSR #5
```

```
                        BCS     ArrayBoundException
                        STR     r1, [Rtmp1, r0, lsl #3]!
                        STR     r2, [Rtmp1, #4]
                        B       State_00000_Interpret
State_01000_do_iadd     ADD     r3, r3, r0
                        B       State_00111_Interpret
State_01000_do_ineg     RSB     r0, r0, #0
                        B       State_01000_Interpret
State_01000_do_land     LDMDB   Rstack!, {r1, r2}
                        AND     R0, R0, R2
                        AND     R3, R3, R1
                        B       State_01000_Interpret

State_01100_Interpret   ...
State_10000_Interpret   ...
State_00101_Interpret   ...
State_01001_Interpret   ...
State_01101_Interpret   ...
State_10001_Interpret   ...
State_00110_Interpret   ...
State_01010_Interpret   ...
State_01110_Interpret   ...
State_10010_Interpret   ...
State_00111_Interpret   ...
State_01011_Interpret   ...
State_01111_Interpret   ...
State_10011_Interpret   ...
```

图 7 展示 Java 字节码指令"laload"，它的功能是从由栈顶位置开始的两个数据字所指定的数据数组中读取两个数据字。从数据数组所读取的这两个字接着替换指定了它们的位置的那两个字以便构成最顶部的堆栈条目。

为了使"laload"指令具有足够的寄存器空间来临时存储那些正从数组中取的堆栈操作数，而不用覆盖指定数组并且在数据数组中定位的输入堆栈操作数，Java 字节码指令被指定具有 2 的需空值，即寄存器存储体内的两个专用于存储堆栈操作数的寄存器在执行 ARM 指令模拟"laload"指令之前必须被清空。如果碰到这个 Java 字节码时，没有两个清空的寄存器，那么可以执行存储操作(STR)以便把当前保持在寄存器中的堆栈操作数压至存储器，从而可以腾出临时存储所需要的空间并且满足该指令的需空值。

当数据位置由数组存储单元和那个数组中的索引指定为两个单独的堆栈操作数时，该指令也有 2 的需满值。附图展示的是，已经满足需满和需空条件并且转换状态为"01001"时的第一状态。"laload"指令分解成三个 ARM 指令。这些指令中的第一个把数组引用加载到寄存器组外部的一个空闲工作寄存器中，它用作堆栈操作数的寄存器高速缓

存。然后，第二个指令使用这个数组引用结合该数组中的索引值，以便存取写到专门存储堆栈操作数的一个空寄存器中的第一数组字。

要极其注意的是，在执行完头两个 ARM 指令之后，系统的变换状态并不改变，并且栈顶指针仍然在其开始的位置，同时所指定是空的寄存器还被那样指定。

ARM 指令序列中的最后指令把第二数组字加载到用于存储堆栈操作数的寄存器组中。因为这是最后的指令，所以如果在它的期间出现了一个中断，那么直到该指令完成之后，该中断才会得到服务，因此可以使用这一指令通过一个变更安全地把输入状态改变成存储堆栈操作数的寄存器的变换状态。在这个示例中，变换状态改变成"01011"，它把新的栈顶指针放在第二个数组字上，并且表示数组引用和索引值的输入变量现在是空寄存器，即给这些寄存器作上“空”的标记就等价于与把它们保持的值从堆栈中移出。

应当注意，尽管"laload"指令的整体堆栈动作并没有改变在这些寄存器中保持的堆栈操作数的数目，不过变换状态的交换却已经出现。在执行最后的操作时实施的变换状态的变更被硬连接到指令翻译机，并且由表示作为"laload"指令特征的“交换参数”来指示，此处该指令翻译机作为正在被翻译的 Java 字节码的函数。

尽管这幅图的示例是一种专门的指令，应当理解的是，所陈述的原理可以扩展到许多不同的 Java 字节码指令，它们模拟 ARM 指令和其它类型的指令。

图 8 是流程示意图，它原理展示上面的技术。在第 10 步，从存储器中取 Java 字节码。在第 12 步，检查用于那个 Java 字节码的需满和需空值。如果需空和需满的任一条件都得不到满足，那么使用第 14 和 16 步，可以执行对堆栈操作数（可能是多个堆栈操作数）分别进行的压入和弹出操作。应该注意的是，这种特殊的系统不允许需空和需满条件同时得不到满足。在第 12 步的条件得到满足之前，可能会需要多次通过第 14 和 16 步。

在第 18 步，选择用于所关心的 Java 字节码的翻译模板中指定了的第一 ARM 指令。在第 20 步，要检查所选择的 ARM 指令是否是在模拟第 10 步中所取的 Java 字节码的过程中要执行的最后指令。如果正在执行的 ARM 指令是最后指令，那么第 21 步用于更新程序计数器的值以

便指向要执行的指令序列中的下一个字节码。应当理解，如果 ARM 指令是最后指令，那么它将会完成它的执行，而不管现在是否出现中断，因此，当系统状态已经达到 Java 字节码的那种匹配正常、没有被中断、完全执行时，就可以安全地更新程序计数器值以便指向下一个 Java 字节码，并重新从那个点开始执行。如果在第 20 步的测试表明还没有到达最后字节码，那么就绕过对程序计数器值的更新。

步骤 22 执行当前 ARM 指令。在第 24 步，要测试是否还有需要按照该模板的部分执行的 ARM 指令。如果还有多条 ARM 指令，那么就在第 26 步选择这些 ARM 指令中的下一个，然后处理就返回到第 20 步。如果不再有指令了，那么处理进入到第 28 步，在该步中，执行指定用于所关心的 Java 字节码的任何变换的改变/交换，以便反映期望的栈顶位置以及包含堆栈操作数的各个寄存器的满/空状态。

图 8 还原理性地展示这样的点，在这些点处，如果有效的中断得到服务，然后中断之后处理重新开始。在进展到第 22 步的当时执行 ARM 指令之后，中断开始得到服务，以不论存储的程序计数器值是多少作为对应字节码序列的返回点。如果执行的当前 ARM 指令是模板序列中的最后指令，那么步骤 21 及时更新程序计数器的值，因此，这将指向下一个 Java 字节码（或者 ARM 指令已经把一个指令集开关初始化）。如果当前执行的 ARM 指令不是序列中的最后指令而是其它，那么程序计数器的值将仍然与在开始执行所关心的 Java 字节码时的那个相同，因此，当返回时，整个 Java 字节码就要重新执行。

图 9 展示 Java 字节码的翻译单元 68，它接收 Java 字节码流并输出翻译的 ARM 指令流（或对应的控制信号）以便控制处理器内核的动作。如前面所描述的，Java 字节码翻译机 68 使用指令模板把简单的 Java 字节码翻译成 ARM 指令或 ARM 指令序列。当每个 Java 字节码已经得到执行，接着减小调度控制逻辑 70 中的计数器值。当这个计数值达到 0 时，然后 Java 字节码翻译单元 68 发出 ARM 指令分支到管理线程或任务间的相应调度的调度代码。

尽管简单 Java 字节码得到由提供了基于高速硬件执行这些字节码的 Java 字节码翻译单元 68 的处理，但是那些需要更复杂处理操作的字节码要送到采用解释例程的形式提供的软件解释器（早已在本说明书给出了所选择的这种例程的示例）。更明确地讲，Java 字节码翻译单

元 68 能够确定：它所接收的字节码是否能得到硬件翻译的支持，因此，根据那个 Java 字节码，要分支到能够找到或引用到用于解释那个字节码的软件例程的地址。这种机制也能够用到当调度逻辑 70 表明需要一个调度操作来产生到调度代码的分支时的情况。

图 10 更详细地展示图 9 的实施方案中的操作以及硬件和软件之间的任务的划分。所有的 Java 字节码由 Java 字节码翻译单元 68 接收，并在第 72 步减小计数器。在第 74 步，要检查计数值是否已经达到 0，如果计数器值已经达到 0（或者从硬连接到系统的预先确定的值向下计数或者可以从用户控制/编程的值向下计数），然后分支到第 76 步的调度代码。一旦在第 76 步完成了调度代码，控制就返回到硬件，处理进入到第 72 步，在此处取下一个 Java 字节码并且再次减小计数器。当计数器达到 0，则它立刻将滚动到新的非零值。备选地，可以把新值强加到计数器中作为第 76 步调度过程的退出部分。

如果在第 74 步的测试表明计数器不等于 0，那么在第 78 步取 Java 字节码。在第 80 步，要判定：所取的字节码是否可在第 82 步用硬件翻译执行，或者所取的字节码是否需要更复杂的处理，以及那样所取的字节码是否应该传递出去用于在第 84 步的软件解释。如果把处理传递出去用于软件解释，那么一旦完成了这步，控制就返回给硬件，此处第 72 步再一次减小计数器以便考虑下取一个 Java 字节码。

图 11 展示备选的控制安排。在第 86 步开始处理时，指令信号（调度信号）置为无效。在第 88 步，要检查所取的 Java 字节码以判定它是否是能得到硬件翻译支持的简单字节码。如果硬件翻译不支持，那么控制就传递出去给解释软件，在第 90 步它执行 ARM 指令例程以便解释 Java 字节码。如果该字节码是得到硬件翻译支持的简单字节码，那么处理就进入第 92 步，在此步骤中，由按照多周期有限状态机形式活动的 Java 字节码翻译单元 68 按照序列发出一个或多个 ARM 指令。一旦 Java 字节码在第 90 步或者在第 92 步已经得到了正确的执行，那么处理就进入到第 94 步，在此步骤中，在第 86 步中把指令信号置为无效之前，在短周期内把它置为有效。对于外部电路来说，把指令信号置为有效表明已经到达了适当的安全点，在该点可能发生基于定时器的调度中断，而不会冒着因为对解释的或翻译的指令的部分执行而损失数据完整性的风险。

图 12 展示电路示例，它可以用于响应在图 11 中所生成的指令信号。在给定的时间周期到期之后，定时器 96 周期性地产生定时信号。这种定时器信号在被清除定时器中断信号清除之前，它一直存储在锁存器 98 中。锁存器 98 的输出与在第 94 步置为有效的指令信号由与门 100 进行逻辑组合。当把锁存器置位并且把指令信号置为有效，然后就生成了作为与门 100 的输出的中断，该中断用于触发使用在系统中为标准中断处理提供的中断处理机制，而执行调度操作的中断。一旦生成了中断信号，这个中断就反过来触发产生清除定时器中断信号，在出现下一个定时器输出脉冲之前，它用于清除锁存器 98。

图 13 是信号示意图，它展示图 12 的电路操作。处理器内核时钟信号以有规律的频率发出。定时器 96 按照预先确定的周期产生定时器信号以便指明当安全的时候应该初始化调度操作。锁存定时器信号。指令信号按照有间隔间距的时间产生，它们要取决于一个特定 Java 字节码以多么快的速度执行。一个简单的 Java 字节码可以在单一的处理器内核时钟周期或更典型的是在两个或 3 个时钟周期内执行，而在一个提供高级管理类型功能的复杂 Java 字节码的执行由软件解释器完成之前，可能要花费几百个处理器时钟周期。无论在哪一种情况下，在表示可以安全地开始调度操作的指令信号发出之前，挂起的置有效位的锁存的定时器信号不触发调度操作。锁存的定时器信号和指令信号的同步出现会触发产生这样的中断信号，它后面立即跟随一个清除锁存器 98 的清除信号。

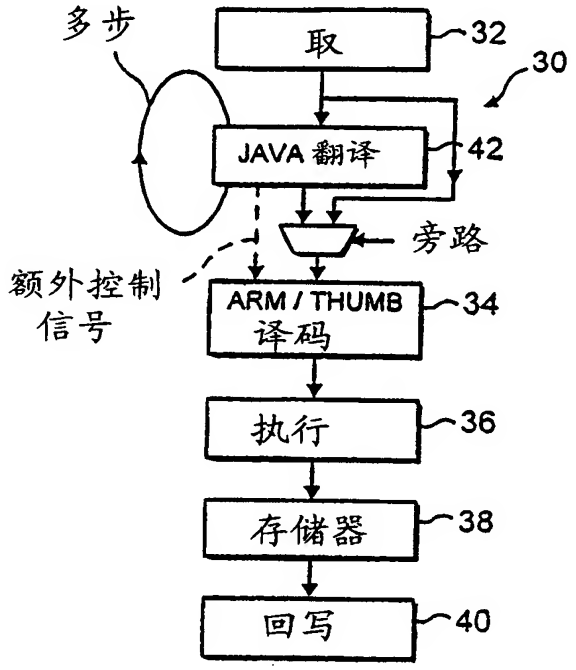

多步
取
32
30
JAVA 翻译
42
旁路
额外控制信号
ARM / THUMB 译码
34
执行
36
存储器
38
回写
40

图 1

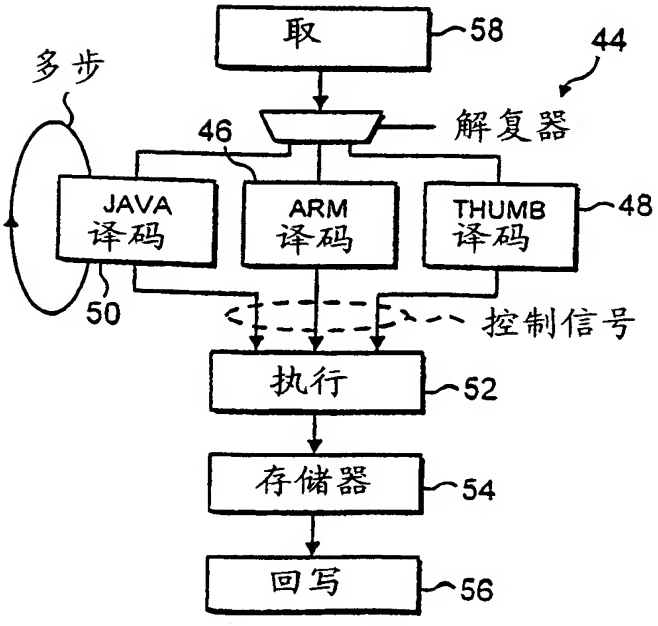

取
58
44
多步
解复器
46
JAVA 译码
ARM 译码
THUMB 译码
48
50
控制信号
执行
52
存储器
54
回写
56

图 2

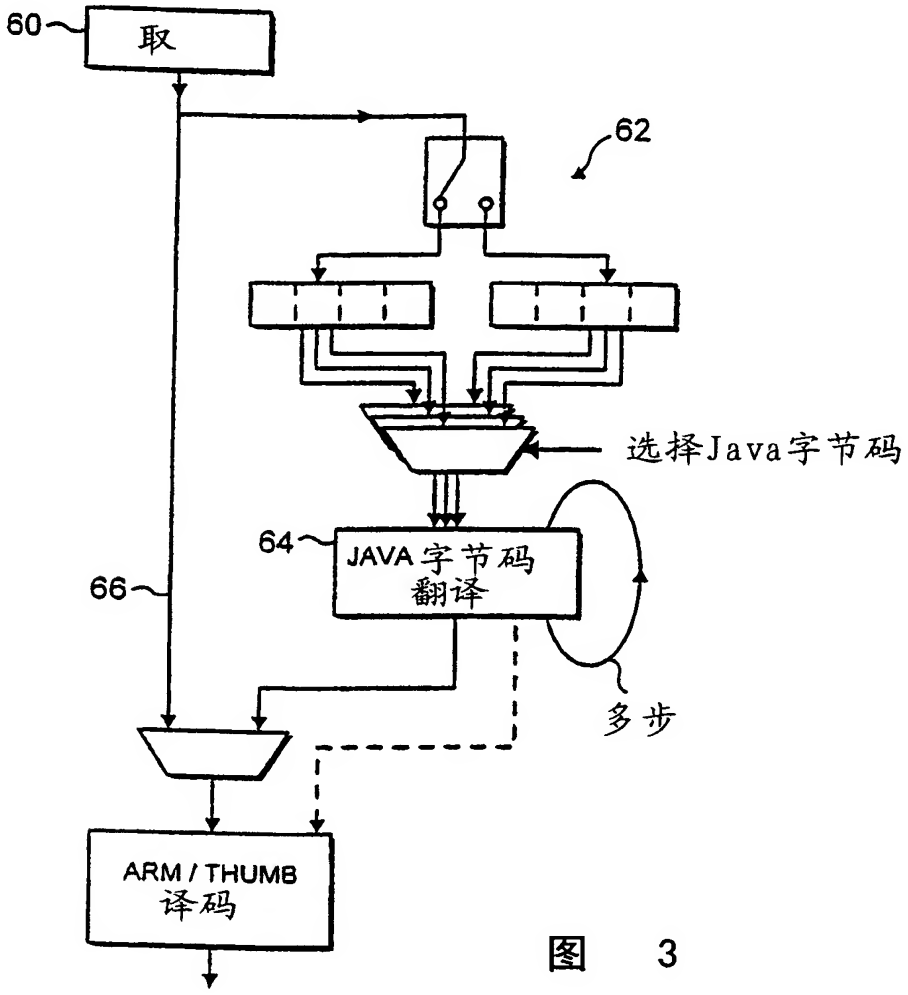

60
取
62
选择Java字节码
64
JAVA 字节码
翻译
66
多步
ARM / THUMB
译码

图 3

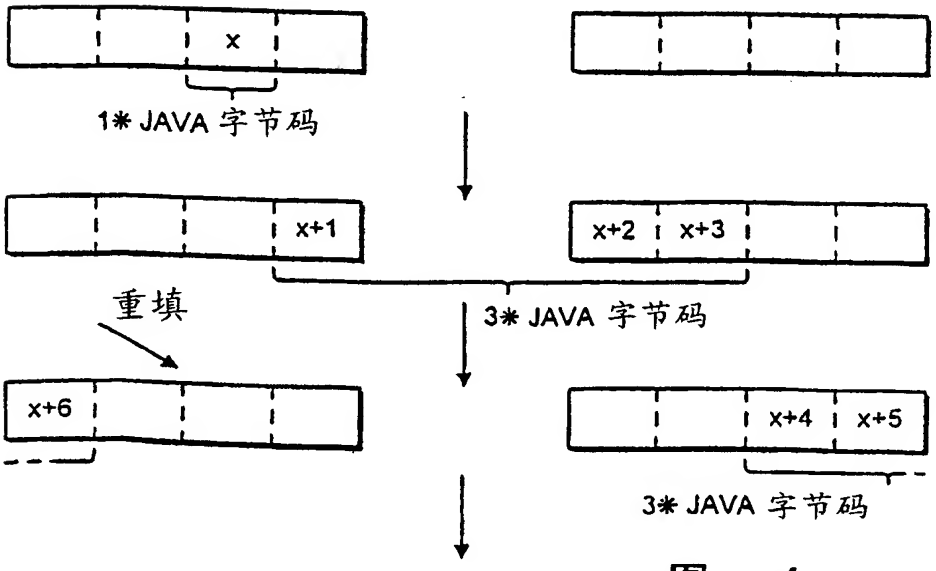

x
1* JAVA 字节码
x+1
x+2
x+3
重填
3* JAVA 字节码
x+6
x+4
x+5
3* JAVA 字节码

图 4

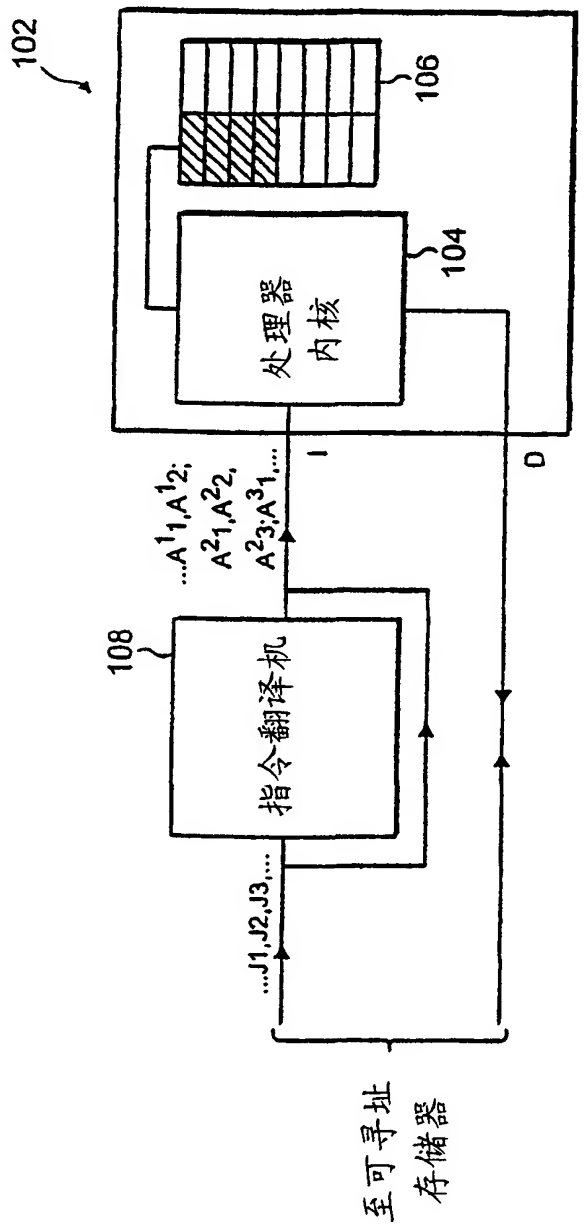

102
106
处理器
内核
104
I
D
...A11,A12;
A21,A22,
A23;A31,...
108
指令翻译机
...J1,J2,J3,...
至可寻址
存储器

图 5

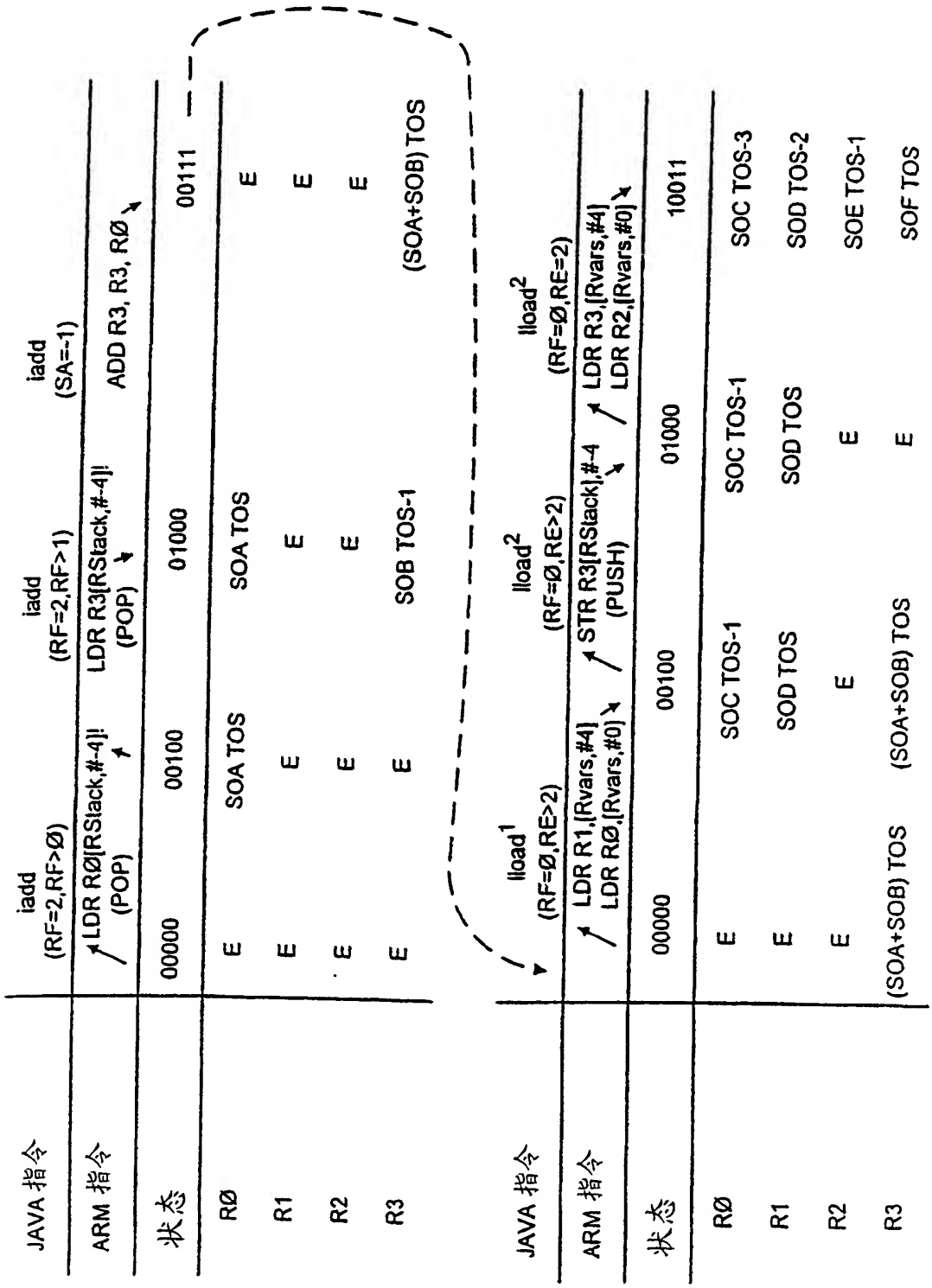

JAVA 指令
iadd (RF=2,RF>Ø)
iadd (RF=2,RF>1)
iadd (SA=-1)
ARM 指令
LDR RØ[RStack,#-4]! (POP)
LDR R3[RStack,#-4]! (POP)
ADD R3, R3, RØ
状态
00000
00100
01000
00111
RØ
R1
R2
R3
E
SOA TOS
SOA TOS
SOB TOS-1
(SOA+SOB) TOS
JAVA 指令
iload1 (RF=Ø,RE>2)
iload2 (RF=Ø,RE>2)
iload2 (RF=Ø,RE=2)
ARM 指令
LDR R1,[Rvars,#4]
LDR RØ,[Rvars,#0]
STR R3[RStack],#-4 (PUSH)
LDR R3,[Rvars,#4]
LDR R2,[Rvars,#0]
状态
00000
00100
01000
10011
RØ
R1
R2
R3
SOC TOS-1
SOD TOS
SOC TOS-3
SOD TOS-2
SOE TOS-1
SOF TOS
(SOA+SOB) TOS

图 6

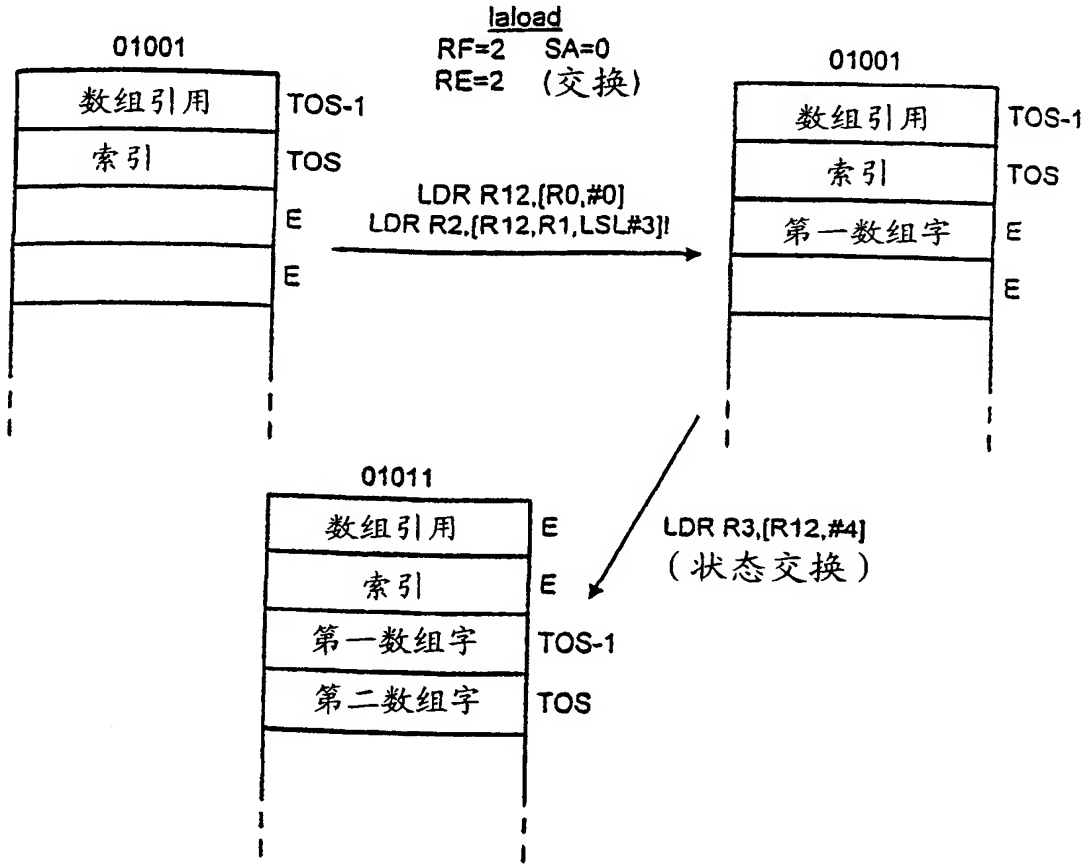

laload
RF=2 SA=0
RE=2 (交换)
01001
数组引用
索引
TOS-1
TOS
E
E
LDR R12,[R0,#0]
LDR R2,[R12,R1,LSL#3]!
01001
数组引用
索引
第一数组字
TOS-1
TOS
E
E
01011
数组引用
索引
第一数组字
第二数组字
E
E
TOS-1
TOS
LDR R3,[R12,#4]
(状态交换)

图 7

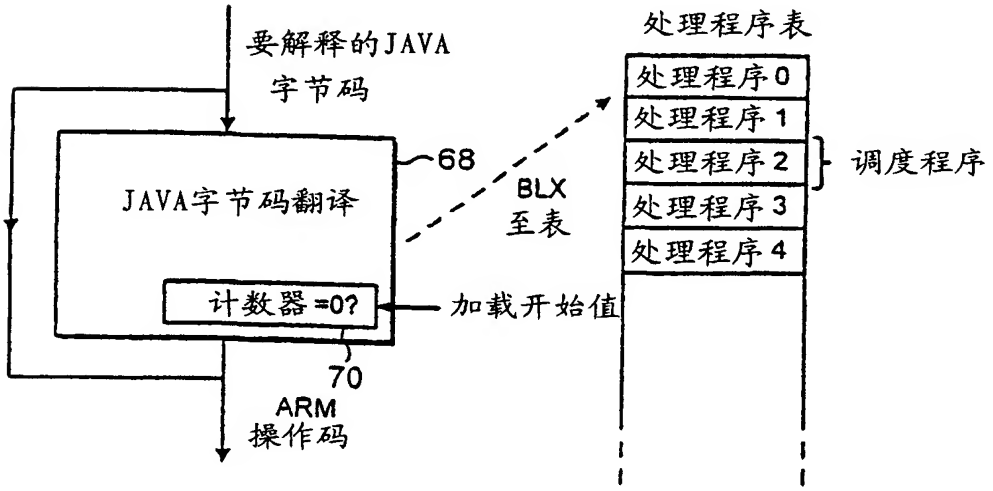

要解释的JAVA
字节码
JAVA字节码翻译
68
BLX
至表
计数器=0?
加载开始值
70
ARM
操作码
处理程序表
处理程序0
处理程序1
处理程序2
处理程序3
处理程序4
调度程序

图 9

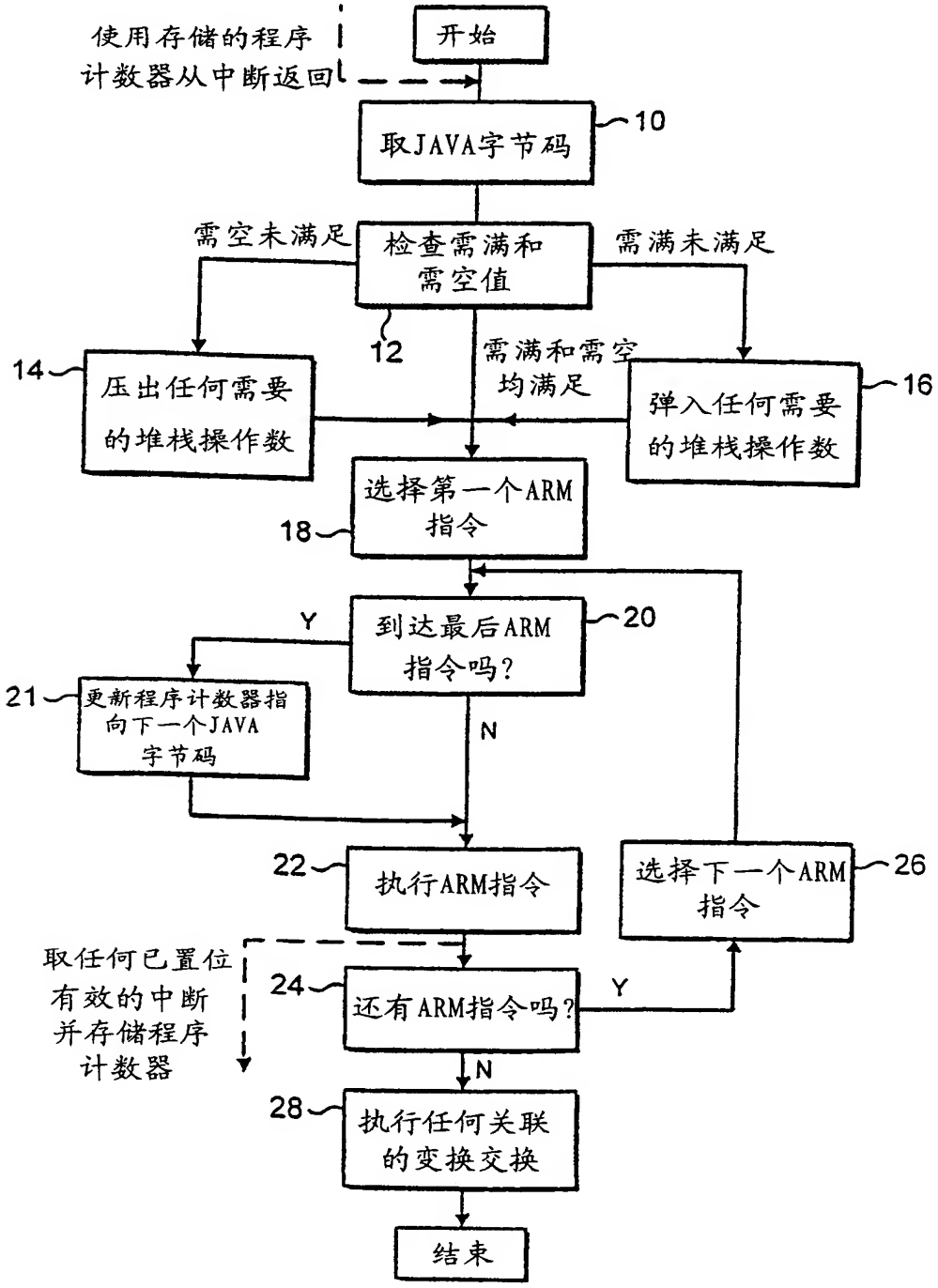

使用存储的程序计数器从中断返回
开始
取JAVA字节码
10
检查需满和需空值
需空未满足
需满未满足
12
需满和需空均满足
14
压出任何需要的堆栈操作数
16
弹入任何需要的堆栈操作数
选择第一个ARM指令
18
到达最后ARM指令吗?
20
Y
N
21
更新程序计数器指向下一个JAVA字节码
22
执行ARM指令
26
选择下一个ARM指令
取任何已置位有效的中断并存储程序计数器
24
还有ARM指令吗?
Y
N
28
执行任何关联的变换交换
结束

图 8

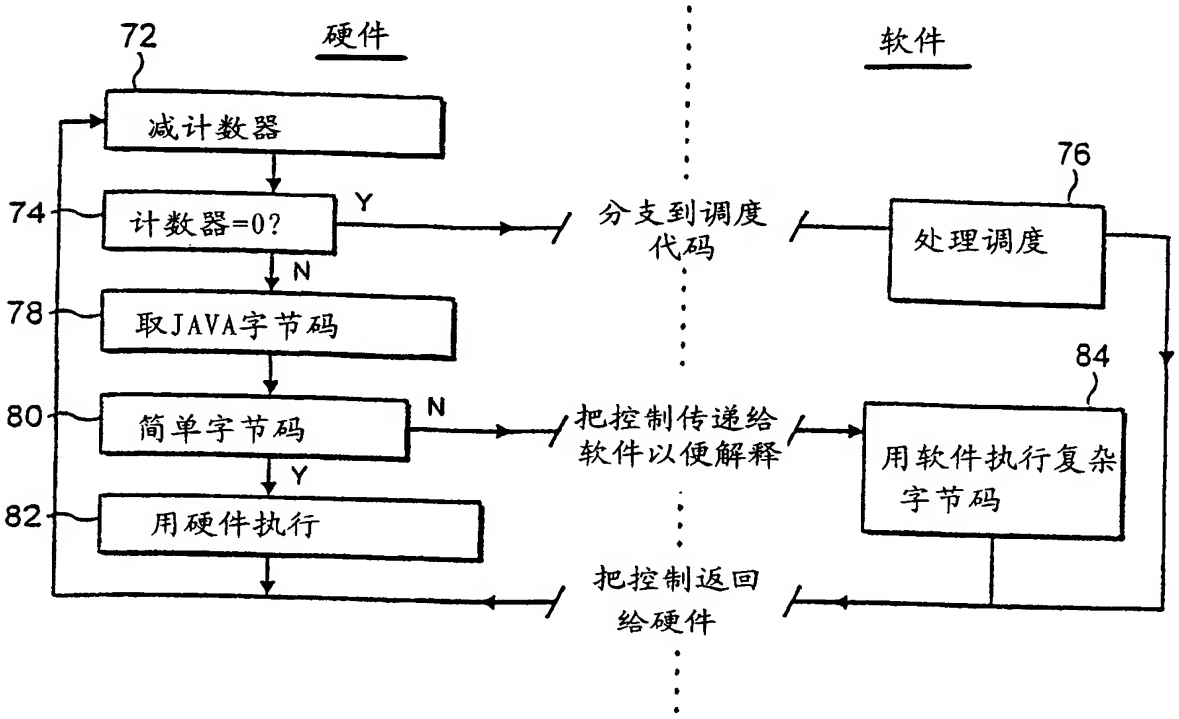

72
硬件
软件
减计数器
74
计数器=0?
Y
N
分支到调度代码
76
处理调度
78
取JAVA字节码
84
80
简单字节码
N
把控制传递给软件以便解释
用软件执行复杂字节码
Y
82
用硬件执行
把控制返回给硬件

图 10

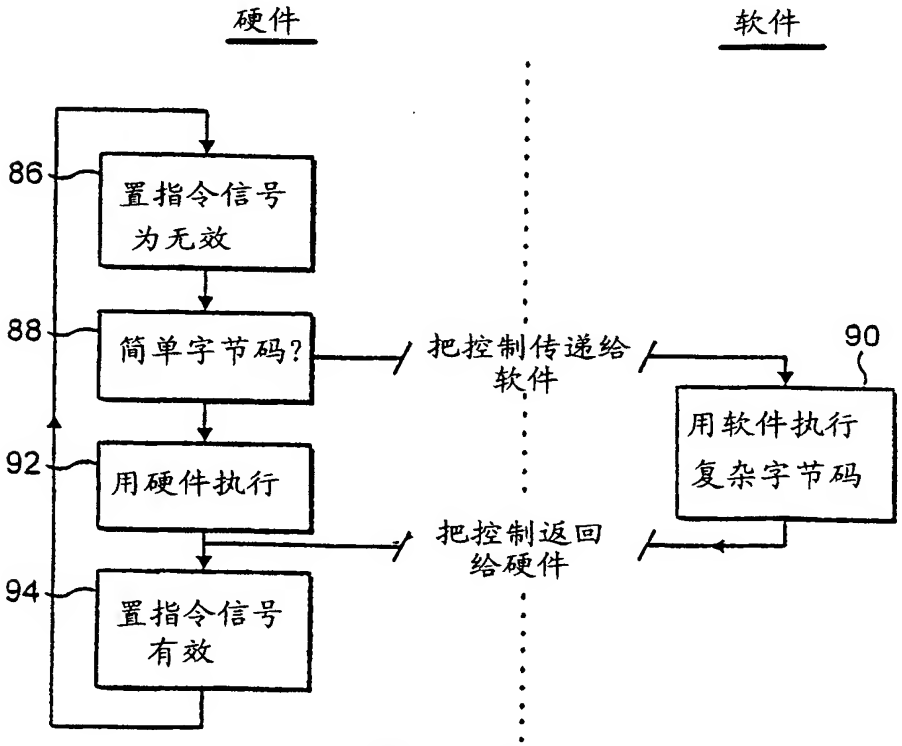

硬件
软件
86
置指令信号为无效
88
简单字节码?
把控制传递给软件
90
用软件执行复杂字节码
92
用硬件执行
把控制返回给硬件
94
置指令信号有效

图 11

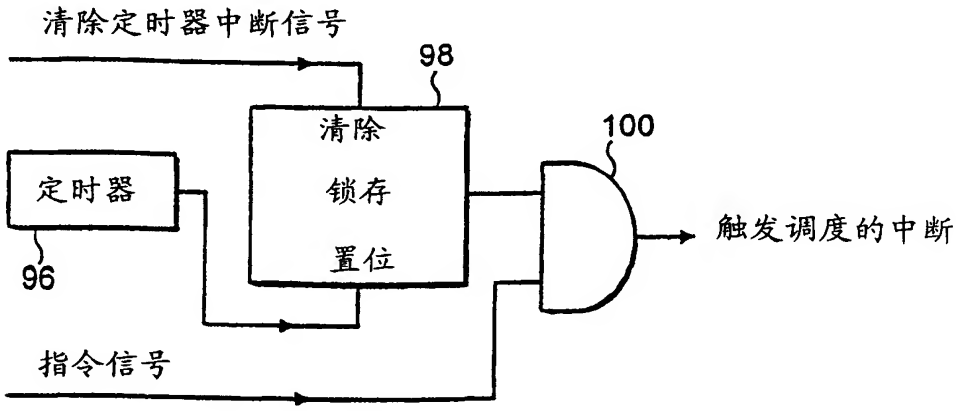

清除定时器中断信号
98
清除
锁存
置位
100
定时器
96
触发调度的中断
指令信号

图 12

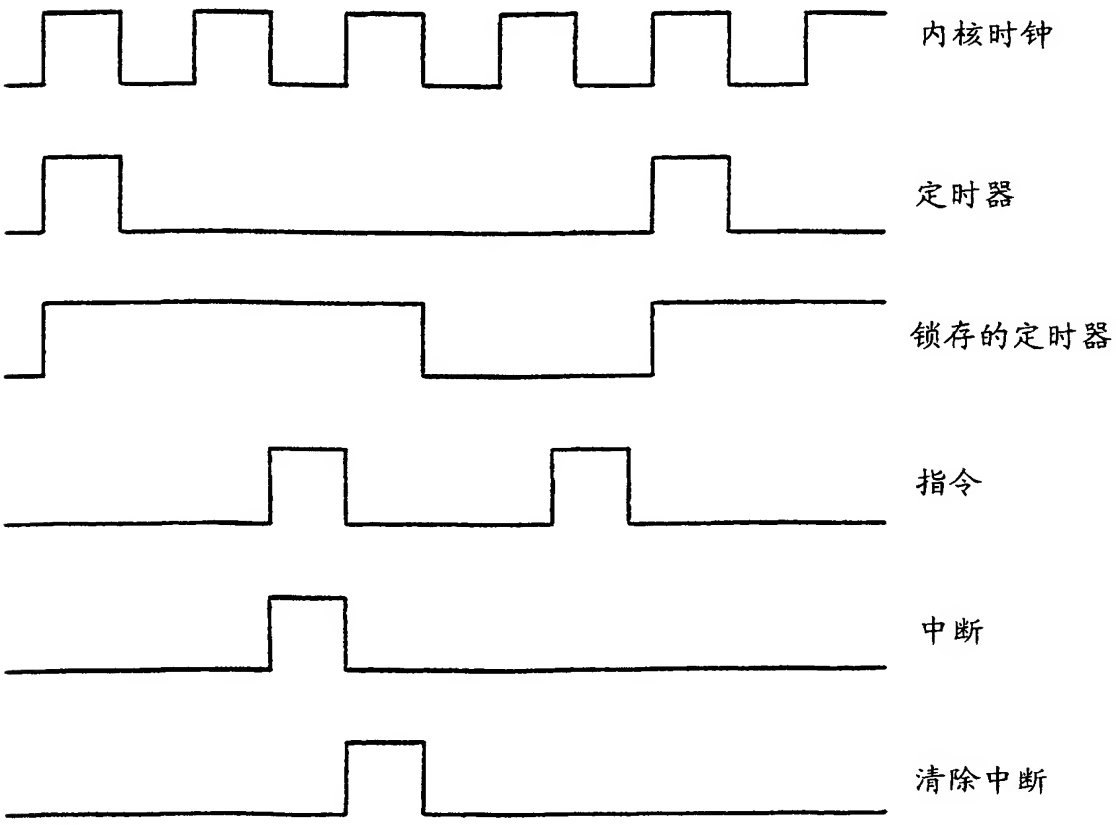

内核时钟
定时器
锁存的定时器
指令
中断
清除中断

图 13